Computer Science Department

TECHNICAL REPORT

A WORST CASE ANALYSIS OF HEAPSORT

BY

CLYDE P. KRUSKAL

AND

ELIA WEIXELBAUM

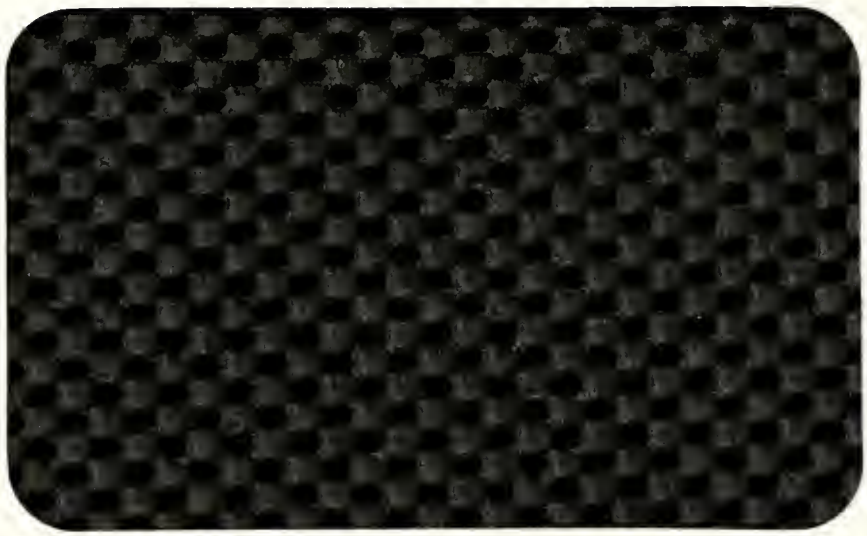
NOVEMBER 1979

REPORT NO. 018

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

esd TR-018
C.2



A WORST CASE ANALYSIS OF HEAPSORT

BY

CLYDE P. KRUSKAL

AND

ELIA WEIXELBAUM

NOVEMBER 1979

REPORT NO. 018

* Submitted May 1977, revised November 1979.

This material is based upon work supported by the Department of Energy under Contract No. DE-AC02-76ER03077.

C.2

ABSTRACT

An exact solution is given for the maximum number of comparisons required by heapsort, assuming that the number of elements to be sorted is one less than a power of two. In addition, an algorithm is presented which produces data yielding the maximum number of comparisons.

ACKNOWLEDGEMENTS: We thank Robert Dewar for suggesting the problem and also for supplying helpful comments and encouragement. In addition, we thank Sal Stolfo and Martin Golumbic for reading an early draft and especially Allan Gottlieb for his careful perusal of umpteen different drafts. Finally, we thank Lucy Jones for her expert typing of this manuscript.

INTRODUCTION:

It is interesting to compare various sorting algorithms based on numbers of comparisons and exchanges. This point is emphasized in Knuth [Kn, sec. 5.3.1]: "... a theoretical study of this subject [counting comparisons] gives us a good deal of useful insight into the nature of sorting processes ..."

The most commonly known 0(n log n) comparison-exchange sorting algorithm not needing external storage is heapsort (sometimes referred to as treesort) [Fl], [Wi]. It is relatively easy to calculate the maximum number of exchanges required by heapsort; in this paper, we calculate the maximum number of (key) comparisons required, assuming that the size of the input is one less than a power of two. In addition, we exhibit an algorithm producing input yielding the maximum number of comparisons.

HEAPSORT ALGORITHM:

DEFINITION: A heap is a binary tree such that the root contains the largest value of the tree, and its sons (if it has sons) are the roots of subtrees that are also heaps.

As usual in heapsort, we view the array, A(1:n) as a tree. The left and right sons of A(i) are A(2i) and A(2i + 1) respectively. There are two phases to heapsort, CREATE_HEAP and SELECT. CREATE_HEAP, as the name indicates, forms a heap from the tree stored in the array, A. SELECT exchanges the root of the tree (which contains the largest value) with the last position of the tree, deletes that last position from the tree, and restores the remainder of the tree into a heap. This procedure is repeated until the root is the only remaining node in the tree. When completed, this results in the array A sorted into increasing order. A simple, unoptimized version of heapsort (figure 1) will be analyzed in this paper.

ANALYSIS OF WORST CASE:

The phases CREATE_HEAP and SELECT will be studied separately. This decision will be justified later.

In this paper, n denotes the number of elements to be sorted, i.e. the number of elements in the tree, and r denotes the number of levels in the tree. The levels of the tree are counted from the top. Therefore, the root is the only element in level 1 and, in general, the L^{th} level contains 2^{L-1} elements. All logs mentioned in this paper are base 2.

```
PROCEDURE HEAPSORT (A,N);
BEGIN
  PROCEDURE SIFT (S,BOUND);
  COMMENT this procedure sifts the item in position S to no lower
                                                    than position BOUND;
    BEGIN
      I := S;   J := 2 * I;   X := A[I];
      WHILE J ≤ BOUND DO
        BEGIN
          IF J < BOUND THEN IF A[J] < A[J+1] THEN J := J + 1;
          IF X ≥ A[J] THEN GOTO DONESIFT;
          A[I] := A[J];   I := J;   J := 2 * I
        END;
DONESIFT:
      A[I] := X
    END SIFT;
  PROCEDURE CREATE_HEAP;
    FOR P := (N DIV 2) TO 1 STEP -1 DO SIFT (P,N);
  PROCEDURE SELECT;
    FOR K := N TO 2 STEP -1 DO
      BEGIN
        HOLD := A[1];   A[1] := A[K];   A[K] := HOLD;
        SIFT (1,K-1)
      END;
  CREATE_HEAP;
  SELECT
END
```

FIGURE 1

For simplicity, this paper considers only those values of n such that the binary tree formed is complete, i.e. $n=2^r - 1$. Furthermore, we can assume that all of the elements are unique since we are seeking the worst case and no additional comparisons could possibly be generated by allowing duplication of elements. Without loss of generality, we also assume the elements to be the integers 1...n.

Since all comparisons are done in SIFT, it follows that in each phase of the algorithm, an upper bound for the worst case occurs if each sift operation forces comparisons to go down to the bottom level. In sifting, there are generally two key comparisons done for every level — one to find the greater son and one to compare the node with that greater son (there is an exception described later which occurs in SELECT). In CREATE_HEAP, we show that this upper bound is achieved. SELECT is more difficult to analyze since, as we will see, for most n, this upper bound is not achieved.

ANALYSIS OF PHASE 1 — CREATE_HEAP:

DEFINITION: A reverse heap is a binary tree such that the root contains the smallest value of the tree, and its sons (if it has sons) are the roots of subtrees that are also reverse heaps.

THEOREM 1: Reverse heaps yield the worst case for CREATE_HEAP. The number of comparisons is $2n - 2\log(n+1)$.

PROOF: When node p is processed, it contains the smallest value of its subtree since it had been the root of a subtree which was a reverse heap and currently has the same descendants simply rearranged. Since node p contains the smallest value, it must sift all the way to the bottom of the tree.

The number of comparisons is counted as follows:
For an element in level L, there are 2 comparisons for each level down to the bottom of the tree. The number of such levels is $r - L$. Since there are 2^{L-1} elements in level L, there are $2^{L-1}(2)(r-L)$ comparisons for the entire level L. CREATE_HEAP proceeds from level r-1 back to level 1. Therefore, the total number of comparisons in the worst case for CREATE_HEAP is $\sum_{L=1}^{r-1} 2^{L-1}(2)(r - L) = 2n - 2 \log(n+1)$. □

THEOREM 2: For every heap, H, there exists a reverse heap, R, such that if CREATE_HEAP were applied to R, H would be created.

REMARK 1: Before proving this theorem, let us examine its significance. When we subsequently analyze SELECT, we will seek a heap that yields the worst case for SELECT. Once this heap is found, theorem 2 shows that it is possible for this heap to have been derived from a reverse heap by CREATE_HEAP. This implies that the number of comparisons in the worst case of heapsort is the sum of the worst cases for CREATE_HEAP and SELECT.

PROOF: We present an algorithm, CREATE_REVERSE_HEAP (figure 2), that, given a heap, produces a reverse heap satisfying the theorem. The algorithm simply reverses the steps of CREATE_HEAP.

```
PROCEDURE CREATE_REVERSE_HEAP(A,N);
  PROCEDURE UNSIFT (S, BOUND);
  COMMENT this procedure unsifts the item in position S up to
          position BOUND;
    BEGIN
      I := S;   J := I DIV 2;   X := A[I];
      WHILE J ≥ BOUND DO
        BEGIN  A[I]  := A[J];   I := J;   J := I DIV 2   END;
      A[I] := X
    END UNSIFT;
  FOR P := 1 TO (N DIV 2) DO UNSIFT(index of node containing
              smallest value in tree rooted by P, P)
```

FIGURE 2

In general, reversing the steps of CREATE_HEAP is a nondeterministic operation; for each step, there are many choices of elements that can be unsifted (i.e. if unsifting at node p, the choices are all of the nodes in the subtree with root p). However, CREATE_REVERSE_HEAP is deterministic since it always unsifts the smallest node in the subtree. In CREATE_REVERSE_HEAP, let T_p denote the tree after the p^{th} call to UNSIFT. Consider the sequence of trees T_0, T_1, T_2 ..., $T_{n \text{ DIV } 2}$. It must be shown the $T_{n \text{ DIV } 2}$ is a reverse heap, and that if CREATE_HEAP were applied to $T_{n \text{ DIV } 2}$, the original heap T_0, would be restored. $T_{n \text{ DIV } 2}$ is clearly a reverse heap since for each node, x, in the tree, there is a call to UNSIFT that moves the smallest node contained in the subtree rooted by x

up to position x, after which descendants of x are merely rearranged amongst themselves. To show the second part, observe that after each call to SIFT in CREATE_HEAP, tree T_i is modified to become T_{i-1}. This is because CREATE_REVERSE_HEAP reverses the order of processing of nodes, and UNSIFT (which is called from CREATE_REVERSE_HEAP) reverses the steps of SIFT. Therefore, CREATE_HEAP, run on $T_{n \text{ DIV } 2}$, creates the original heap, T_0. □

As an example, consider CREATE_REVERSE_HEAP applied to the heap ${}_4 6_3 {}^7 {}_2 5_1$. On processing node 1 (which contains the value 7), the 1 gets unsifted from the bottom to the top yielding ${}_4 6_3 {}^1 {}_2 7_5$. On processing node 2, the 3 gets unsifted yielding ${}_4 3_6 {}^1 {}_2 7_5$. Finally, on processing node 3, the 2 gets unsifted giving us the reverse heap: ${}_4 3_6 {}^1 {}_7 2_5$. If this sequence of data were run through CREATE_HEAP, it would produce the original heap, ${}_4 6_3 {}^7 {}_2 5_1$.

ANALYSIS OF PHASE 2 - SELECT:

We begin this section by giving a upper bound for the worst case for SELECT. As in CREATE_HEAP, this would happen if every sift operation forced the element being sifted to end up in the bottom level — or next to bottom level provided that descendants of that position are in the tree (the reason the next to bottom level is allowed is that the comparison will still have to be done to determine if the item should fall to the bottom level). However, more care is needed in calculating the upper bound here since the tree size is diminishing. In SELECT, the root element is switched with the last element of the tree, then the last node with its new element (the old root) is deleted from the tree and the (new) root element is sifted. This process is repeated until there is one node left in the tree.

Assume that the root element has just been switched with an element in level L and the node associated with this element in level L has been deleted. A sift operation on the (new) root element must now be done. There are three cases to consider:

1) After deletion, there are no elements remaining in level L. This occurs once in processing level L. In this case, the farthest point to which the root can be sifted is to level L - 1. Therefore, the maximum number of comparisons here is 2(L - 2).

2) After deletion, one item remains in level L. This also occurs once in processing level L. In this case, the farthest point to which the root can be sifted is the single spot in level L. To get to level L - 1, there are 2(L - 2) comparisons and then one more comparison is done to determine if the item should be sifted into level L (only one comparison since there is no brother to compare to).

3) After deletion, there is more than one element remaining in level L. This occurs $2^{L-1}-2$ times in processing level L. There are 2(L - 1) comparisons to sift the element to the bottom level. When the item being sifted reaches the bottom level, it must go to a position that has a brother, otherwise the maximum number of comparisons will not be achieved.

Summing over the 3 cases for L yields

$$2(L-2)+(2(L-2)+1)+(2^{L-1}-2)2(L-1) = L2^L - 2^L - 3$$

comparisons. We will refer to this upper bound as ub_L. This is an upper bound on the number of comparisons in running SELECT on level L. SELECT proceeds from level r back to level 2. Therefore, an upper bound on the number of comparisons for SELECT is

$$\sum_{L=2}^{r} (L2^L - 2^L - 3) = \sum_{L=2}^{r} ub_L = 2n \log(n+1) - 4n - \log(n+1) + 3.$$

We will refer to this upper bound as UB_n and any heap achieving this bound as a UB_n-heap (where n is the number of nodes in the tree). Now, we must determine how close to UB_n the actual worst case is.

Just as we found the worst case of CREATE_HEAP by reversing the steps of the algorithm, we try the same method now. Given a permutation of 1, 2, ..., $2^{L-1}-1$ that yields the worst case for a tree containing L - 1 levels, make this into a tree containing L levels by assigning values 2^{L-1} through $2^{L} - 1$ to nodes 2^{L-1} through $2^{L} - 1$ respectively. This is an assignment of data that an L-level tree might have after having run SELECT for all elements in level L. The entire L^{th} level is considered to be currently deleted from the tree. Now reverse the steps of SELECT on this L^{th} level to produce a worst case heap for L levels. For each element, x, in level L, proceeding from left to right, unsift an element, y, to the root, switch the root, y, with x and consider the position now containing y to be in the tree.

REMARK 2: Any UB_{2L+1-1}-heap (L+1 levels) can be obtained by unsifting a UB_{2L-1}-heap (L levels). In fact, only a UB_{2L-1}-heap can be unsifted to a UB_{2L+1-1}- heap.

Upon sifting, we must be sure to choose an appropriate element so that when the root is switched with the deleted element in the bottom level, the element will be small enough to preserve a heap. Furthermore, in order to achieve the upper bound described, elements must be unsifted from one of two possible positions: 1) the bottom level (the bottom is level L - 1 the first time and level L the remainder of the time), where the node in that position has a brother that is not deleted from the tree, or 2) the next to the bottom level, where the node in that position has two sons, both of which

are not deleted from the tree.* An exception occurs when there is only one node in the bottom level. In this case, simply unsift that node (or its father) regardless of the fact that it does not have a brother (or two sons) in the tree.

EXAMPLE: The trees $_2{}^3{}_1$ and $_1{}^3{}_2$ are both heaps that yield UB_3 comparisons and therefore UB_3 is the least upper bound (i.e. the worst case). The sequence of trees in figure 3 demonstrates the steps being reversed to get from $_2{}^3{}_1$ to the 3-level worst-case heap $_5{}^6{}_4{}^7{}_1{}^3{}_2$. Thus, UB_7 is also the least upper bound. The circled numbers refer to the node being unsifted, the dotted circles represent alternative choices that also lead to worst-case heaps and the cut-off portions of the trees are the nodes currently deleted from the tree. Observe that in the starred tree, had the 1 been chosen for unsifting, then upon going forward, when the 1 would have been sifted, there would have been only one comparison at the bottom level since the node where the 1 lies does not have a brother. Had the 4 been unsifted in the starred tree, the following sequence would have resulted: $_2{}^6{}_3{}^4{}_1{}^3{}_7 \rightarrow {}_2{}^6{}_5{}^7{}_1{}^3{}_4$. However, the second tree is not a heap.

* Of course, if a node in the next to the bottom level would be small enough to preserve a heap, then both of its sons would also since they are even smaller.

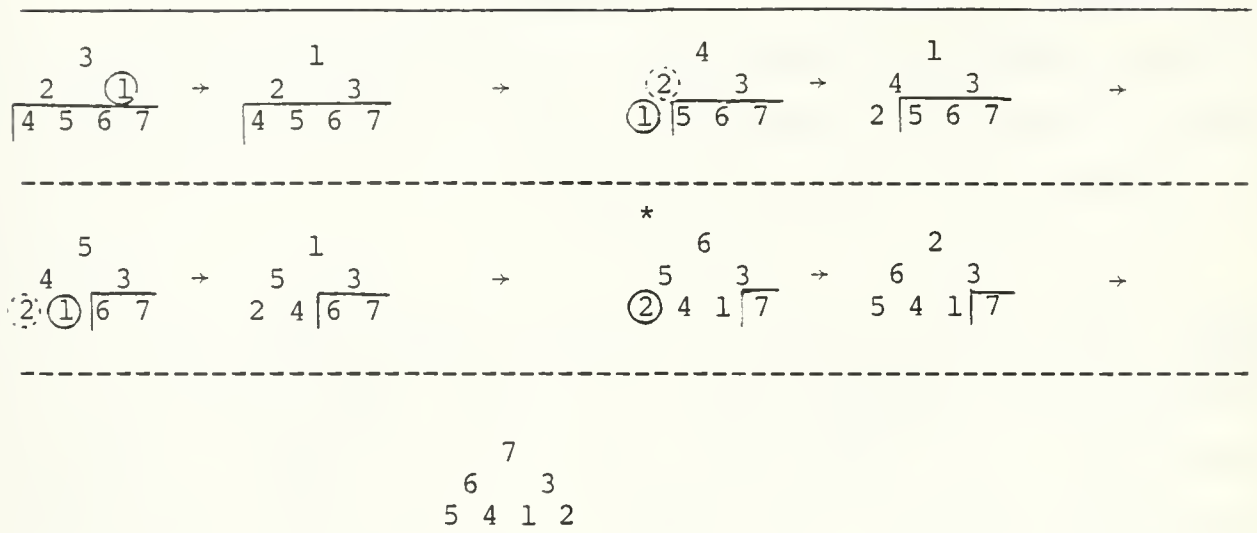


FIGURE 3

By considering all other possible valid choices for unsifting starting with $\begin{smallmatrix} & 3 \\ 2 & 1 \end{smallmatrix}$, only the following UB_7-heaps can be obtained:

$$\begin{smallmatrix} & 7 \\ 6 & & 3 \\ 5 & 4 & 2 & 1 \end{smallmatrix} \quad , \quad \begin{smallmatrix} & 7 \\ 6 & & 3 \\ 4 & 5 & 1 & 2 \end{smallmatrix} \quad , \text{ and } \quad \begin{smallmatrix} & 7 \\ 6 & & 3 \\ 4 & 5 & 2 & 1 \end{smallmatrix} \; .$$

No UB_7-heap can be obtained from $\begin{smallmatrix} & 3 \\ 1 & & 2 \end{smallmatrix}$ by unsifting according to the method immediately preceding the last example, as simple data manipulation verifies. By that same method, we could start with any of the four UB_7-heaps and attempt to obtain a UB_{15}-heap. Unfortunately, this does not work as a position is obtained in which no possible node exists in the bottom or next to bottom level for unsifting as previously described. This implies that no UB_n-heap exists where $n=2^r-1$ and $r \geq 4$ (see remark 2).

We now present a series of lemmas which show by exactly how many comparisons UB_n exceeds the number of comparisons in the worst case.

LEMMA 1: If the element 2 does not begin in the leftmost position of the bottom level of an L-1 level heap as the steps are reversed from level L - 1 to level L, then an L-level heap requiring ub_L comparisons for the L^{th} level cannot be constructed.

PROOF: Assume that as we begin to reverse the steps from level L-1 to level L, the 2 starts out anywhere but the leftmost position of level L-1. We must immediately choose the 1 for unsifting, for if we do not, then after some unsift operations, the 1 would be the father of the next node to be restored to the tree. Therefore, there would be no possible element to be unsifted since any potential choice would be greater than 1 and would have to be switched to the position under the 1, creating a tree that is not a heap (i.e. a tree which is not reachable at this point in SELECT).

However, if the 1 is the first element unsifted, then the 2 cannot be chosen after that, without a loss of comparisons from ub_L: As with the 1, after some unsift operations, the 2 would be the father of the next node to be restored to the tree. In this case, only the 1 could be unsifted and then switched to the bottom level, becoming the left son of the 2. The next node to be restored to the tree would be the right son of the 2. Unfortunately, the only elements that could be unsifted and switched into that position would be the 1, which currently has no brothers, or the 2 which only has one son (the 1). □

LEMMA 2: If at some point in backing up from level L - 1 to level L, the 1 and the 2 are on level L, the remainder of the backing up can be completed without losing any (more) comparisons from ub_L.

PROOF: The 1 can be unsifted if it has a brother in level L. Since the 1 is unsifted from the bottom level no comparisons are lost. If it does not have a brother, then the 2 can be unsifted, also from the bottom level. In either case, the 1 or 2 is small enough to preserve a heap, since the smallest element in level L - 1 is bigger than 2. □

LEMMA 3: If the 2 begins in the leftmost position of level L - 1 as the steps are reversed in backing up to level L, then level L can be added so that ub_L comparisons are required for that level. Furthermore, if this is to be accomplished, then it is impossible for the 2 to remain in the leftmost position of level L.

PROOF: Unsift the 1 from level L - 1 and switch it with the first element in level L. The 1 is now the left son of the 2 and it is the only element in level L. Unsift the 1 again (unsifting the 2 would work also). Upon unsifting the 1, the 2 falls into level L. The 1 gets switched with the next position in level L. So far, no comparisons have been lost and by lemma 2, the remainder of the backing up can be completed without forcing any loss of comparisons on this level. Figure 3 (page 12) demonstrates this method.

Assume now that there were a method that would result in a heap needing ub_L comparisons for the L^{th} level and containing the 2 in the leftmost position of level L. Any method that works must unsift the 1 for the first time resulting in the tree

in figure 4a, and then again resulting in the tree in figure 4b. After the 1 is picked the first time, the 3 must be in level L - 1, but not in the leftmost position, since the 2 is still there. Later, when the 1 is eventually unsifted and switched under the 3 (figure 4c), nothing other than the 1 will be able to be switched under the 3, excluding the 2, which by assumption must be left in the leftmost position of level L. The 1 would have to be unsifted again at a loss of one comparison from ub_L. □

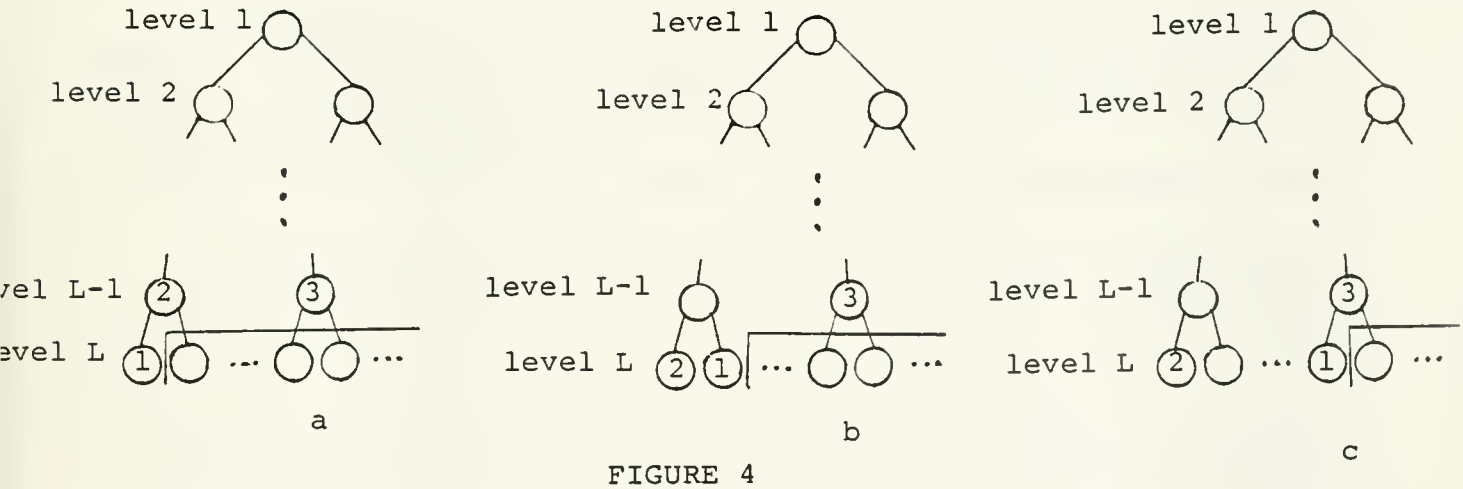


FIGURE 4

LEMMA 4: In backing up from level L - 1 to level L, it is only necessary to lose at most one comparison from ub_L.

PROOF: If the 2 begins in the leftmost position, we know from lemma 3 that the backing up can be done without forcing a loss of any comparisons. Therefore, we may assume that the 2 does not begin in the leftmost position. Begin by unsifting the 1 from level L - 1 and switch it with the first position in level L. Now, instead of unsifting the 1 again, unsift the 2 from level L - 1 and switch it to the next position in level L. This causes a loss of one comparison from ub_L. However, now the

1 and the 2 are both in level L and therefore, by lemma 2, the remainder of the backing up can be completed without forcing any further loss of comparisons from ub_L. □

We will refer to lb_L as the following lower bound on the number of comparisons in the worst case for running SELECT on level L:

$$lb_L = \begin{cases} ub_L & \text{if } L \leq 3 \\ ub_L - 1 & \text{otherwise} \end{cases}$$

At this point, we present a lower bound for the worst case of SELECT by giving an algorithm that generates heaps that yield this lower bound in comparisons. Recall that $UB_n = 2_n \log(n+1) - 4n - \log(n+1) + 3$ is the upper bound for the worst case number of comparisons in SELECT.

THEOREM 3: A lower bound, LB_n, for the worst case of comparisons in SELECT is

$$LB_n = \sum_{L=2}^{r} lb_L = \begin{cases} UB_n & = 2n \log(n+1) - 4n - \log(n+1) + 3 \text{ if } n \leq 7 \\ UB_n - (\log(n+1) - 3) & = 2n \log(n+1) - 4n - 2\log(n+1) + 6 \text{ otherwise.} \end{cases}$$

PROOF: The algorithm, REVERSE_SELECT, in figure 5 demonstrates how these heaps are created. If n = 1, 3, or 7 (r = 1,2, or 3), the lower bound is the exact worst case as we have already shown. Since none of the possible worst-case three-level heaps has the 2 in the leftmost position of level three, by lemmas 1 and 4, we can back up to level four with a loss of one comparison from ub_4 and consequently from UB_{15}. REVERSE_SELECT uses the method described in the proof of lemma 4 and, therefore, upon completion of backing

up from level L - 1 to level L, the 2 will not be in the leftmost position of level L. Thus, for $L \geq 4$, SELECT will take $ub_L - 1$ comparisons from level L to level L - 1.

Therefore, if $n>7$ $(r>3)$, the number of comparisons done on the heaps created by this algorithm is $\log(n+1) - 3$ less than UB_n, i.e. if $n>7$ the number of comparisons is

$$LB_n = \sum_{L=2}^{r} lb_L = ub_2 + ub_3 + \sum_{L=4}^{r} (ub_L - 1). \qquad \square$$

The remainder of this paper is devoted to showing that the lower bound for the worst case in comparisons given in theorem 3 is in fact the exact worst case.

In REVERSE_SELECT, from the point that backing up commences from level three to level four until termination of the algorithm, the 2 never begins in the leftmost position of level L - 1. This is the reason that the lower bound is obtained by subtracting one comparison from UB_n for every level after the third. However, one might conjecture that there are trickier methods of constructing heaps so that less than one comparison is lost for every level — perhaps one comparison for every other level. The only way to do this would be to make sure that for some L, the 2 ends up in the leftmost position of level L - 1, without loss of comparisons, since it is known from lemma 3 that in this case, backing up can be done from level L without forcing any loss of comparisons from ub_L. This, as we will see now, cannot be done.

```
PROCEDURE REVERSE_SELECT;
  IF N = 1 THEN A[1] := 1
  ELSE IF N = 3 THEN
    BEGIN
      A[1] := 3;  A[2] := 2;  A[3] := 1
    END
  ELSE
    BEGIN
      A[1] := 7;  A[2] := 6;  A[3] := 3;  A[4] := 5;
           A[5] := 4;  A[6] := 1;  A[7] := 2;
      FOR L := 4 TO LOG(N+1) DO
        FOR K := 2^(L-1) TO 2^L - 2 STEP 2 DO
          BEGIN
            UNSIFT(index of node containing the 1, 1);
            A[1] := K;
            A[K] := 1;
            UNSIFT(index of node containing the 2, 1);
            A[1] := K+1;
            A[K+1] := 2
          END
END
```

FIGURE 5

LEMMA 5: If, after completion of backing up to level L, $L \geq 3$, the 2 is in the leftmost position of level L, then, for some k, $3 \leq k \leq L$, level k must take at most $lb_k - 1$ comparisons for SELECT and at most $lb_{k'}$ comparisons for k' = k+1 to L (figure 6).

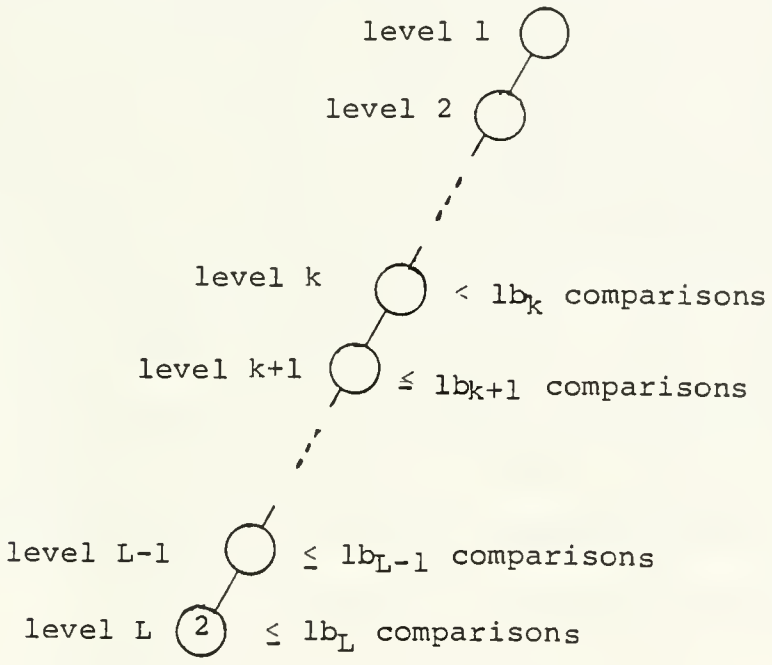


FIGURE 6

PROOF: We assume L >3 as the truth of the lemma can easily be verified for L = 3.

CASE 1: Assume that, when starting the backup from level L - 1 to level L, the 2 is not in the leftmost position of level L - 1. The only way to get it to the leftmost position of level L is to unsift it first. The 2 cannot be unsifted anymore during the processing of this level, since it would not be able to remain in the leftmost position of level L. If the 1 is not unsifted next then it will have to be unsifted later from level L-1, at a loss of two comparisons from ub_L,

i.e. one comparison from lb_L. Therefore, assume the 1 is unsifted now. This is at a loss of one comparison from lb_L. If the father of the 1, after the 1 is switched to level L, is not the 3, then the 3 lies elsewhere in level L-1 (figure 4b, page 15), and when the 1 is eventually unsifted and switched under the 3 (figure 4c), nothing other than the 1 will be able to be switched under the 3, excluding the 2, which must be left in the leftmost position of level L. The 1 would have to be unsifted again but that would be at a loss of another comparison from ub_L, i.e. one comparison from lb_L. Therefore, we may assume that the 3 is the father of the 1 after the 1 is switched to level L (figure 7). By now alternating between the 1 and the 3 as elements for unsifting, the remainder of the level can be processed without forcing any further loss of comparisons from ub_L, i.e. none from lb_L.

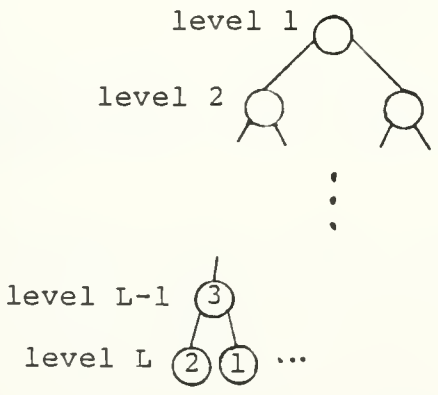


FIGURE 7

The question now is how the 3 could have gotten into the leftmost position of level L-1. There are two cases to consider:

A) Before beginning to back up to level L, the 3 was in the leftmost position of level L-2 which means only the 1 and 2 could be under it, figures 8a and 8b. The situation in figure 8a cannot occur since we are assuming the 2 is not in the leftmost position of level L-1. As for the situation in figure 8b, at the beginning of backing up to level L had the 1 been chosen first for unsifting, the 3 would have fallen into the leftmost position of level L-1. However, we are assuming that the two had been chosen first, forcing the 3 away from the leftmost position of level L-1. Thus, this case cannot occur.

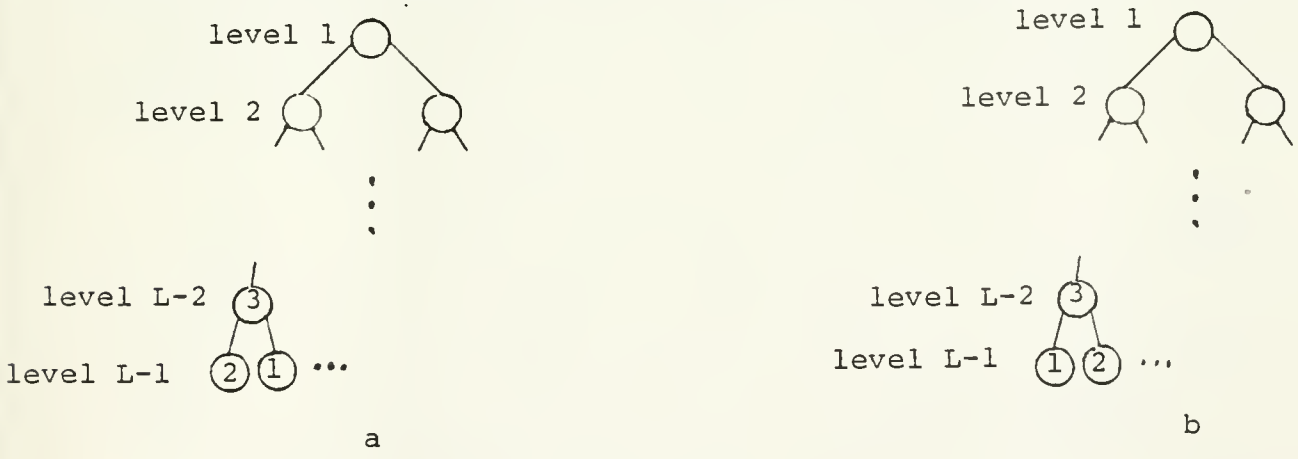


FIGURE 8

B) Before beginning to back up to level L, the 3 was in the leftmost position of level L-1. There are two subcases:

i) The 3 gets put into that position from level L-2 by being the first element unsifted in backing up to level L-1. When the 1 and 2 are later unsifted from level L-2, two comparisons must be lost from ub_{L-1}, i.e. at least one from lb_{L-1}.

ii) The 3 falls into that position. This can only happen if, at some point, the 3 had been in the leftmost position of level L - 2. Assuming that the 3 is in the leftmost position of levels L-1, L-2, L-3,..., we lose at least one comparison from each of ub_{L-1}, ub_{L-2}, ub_{L-3}, ..., i.e. we do not add any comparisons to lb_{L-1}, lb_{L-2}, lb_{L-3}, ... since the 3 and not the 2 is in the bottom leftmost position. If for some k>3, the 3 is not in the leftmost position of level k-1, case i applies (for L = k+1) forcing a loss of one comparison from lb_k. Otherwise, we reach level three with the 3 in the leftmost position. Thus, the three-level heap is not a UB_7-heap, meaning that at least one comparison must have been lost from ub_3 and therefore from lb_3.

CASE 2: Assume the 2 is in the leftmost position of level L-1. By lemma 3, there must be at least a one comparison loss from ub_L to keep the 2 in the leftmost position of level L. Assuming that the 2 is in the leftmost position of levels L, L-1, L-2, ..., we lose at least one comparison from ub_L, ub_{L-1}, ub_{L-2}, ... by applying lemma 3 (to each level). If for some k>3, the 2 is not in the leftmost position of level k-1, case 1 applies (for L = k). Otherwise, the 2 remains in the leftmost position of level 3 satisfying the case with k=3 (cf. case 1-B-ii). □

THEOREM 4: The number of comparisons in the worst case of heapsort is

$2n \log(n+1) - 2n - 3 \log(n+1) + 3$ if $n \leq 7$ and

$2n \log(n+1) - 2n - 4 \log(n+1) + 6$ if $n > 7$.

PROOF: From theorem 3, we know that a lower bound for the worst case gives up one comparison from ub_L for every level $L>3$. From lemma 5, we see that any attempt to improve this lower bound fails since in setting up a level that can be processed without any loss of comparisons, an earlier level must first be set up requiring a loss of too many comparisons. Therefore, the lower bound given in theorem 3 represents the exact worst case for SELECT. The worst case for the entire algorithm is obtained by adding the values in theorems 1 and 3 as we are permitted to do on the basis of remark 1. □

APPENDIX

```
PROCEDURE WORST_CASE_OF_HEAPSORT (A,N);
COMMENT this algorithm produces an array, A, that yields the worst
  case number of comparisons for heapsort, assuming that N is 1
  less than a power of 2;
BEGIN
  PROCEDURE UNSIFT (S, BOUND);
  COMMENT this procedure unsifts the item in position S up to
    position BOUND;
    BEGIN
      I := S;  J := I DIV 2;  X := A[I];
      WHILE J ≥ BOUND DO
        BEGIN A[I] := A[J];  I := J;  J := I DIV 2  END;
      A[I] := X
    END;
  PROCEDURE REVERSE_SELECT;
    IF N = 1 THEN A[1] := 1
    ELSE IF N = 3 THEN
      BEGIN  A[1] := 3;  A[2] := 2;  A[3] := 1  END
    ELSE
      BEGIN
        A[1] := 7;  A[2] := 6;  A[3] := 3;  A[4] := 5;
             A[5] := 4;  A[6] := 1;  A[7] := 2;
        FOR L := 4 TO LOG(N+1) DO
          FOR K := 2^(L-1) TO 2^L - 2 STEP 2 DO
            BEGIN
              UNSIFT (index of node containing the 1, 1);
              A[1] := K;  A[K] := 1;
              UNSIFT (index of node containg the 2, 1);
              A[1] := K+1;  A[K+1] := 2
            END
      END;
PROCEDURE CREATE_REVERSE_HEAP;
   FOR P := 1 TO (N DIV 2) DO UNSIFT(index of node containing smallest
                                     element in tree rooted by P, P);
 REVERSE_SELECT;
 CREATE_REVERSE_HEAP
END
```

BIBLIOGRAPHY

[Er] Erkiö, H., "On heapsort and its dependence on input data," Report A-1979-1, Dept. of Computer Science, University of Helsinki, Finland, 1979.

[Fl] Floyd, R.W., "Treesort 3: Algorithm 245," Communications of the ACM, 7, 12(Dec. 1964), 701.

[Kn] Knuth, D.E., The Art of Computer Programming, Vol 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

[Wi] Williams, J.W.J.,"Heapsort: Algorithm 232," Communications of the ACM, 7, 6(June 1964), 347-348.

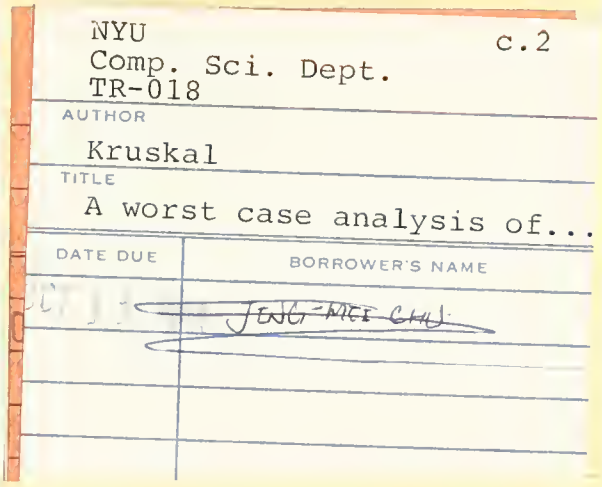
This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

OCT 1 1 1984			
GAYLORD 142			PRINTED IN U.S.A.

NYU
Comp. Sci. Dept.
TR-018
Kruskal
A worst case analysis of ...
c.2


NYU
Comp. Sci. Dept.
TR-018
c.2
AUTHOR
Kruskal
TITLE
A worst case analysis of...
DATE DUE
BORROWER'S NAME
JENG-MEI CHU

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012